Muhammad Dzaky Farhan
5025211069

1.

1. False
2. True
3. False
4. False
5. False
6. True
7. True
8. False
9. True
10. True
11. False
12. True
13. False
14. False
15. True

2. A possible polynomial-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers is the dynamic programming approach. This approach involves solving the problem recursively, by finding the solution to smaller subproblems and then combining these solutions to obtain the solution to the original problem.
Here is the pseudo-code for this algorithm:

```
function LIS(seq):
  # Create an array to store the length of the longest increasing subsequence ending at each index
  L = array of length n, initialized to 0

  # Initialize the length of the longest increasing subsequence ending at the first index to 1
  L[0] = 1

  # Loop through the sequence starting from the second element
  for i = 1 to n-1:
    # Set the length of the longest increasing subsequence ending at the current index to 1
    L[i] = 1

    # Loop through the sequence from the beginning to the current index
    for j = 0 to i-1:
      # If the current element is greater than the element at the previous index
      # and the length of the longest increasing subsequence ending at the previous index is greater than the current length
      if seq[i] > seq[j] and L[j] > L[i]:
```

```
        # Update the length of the longest increasing subsequence ending at the current index
        L[i] = L[j] + 1

    # Return the maximum value in the L array
    return max(L)
```

The time complexity of this algorithm is O(n^2), since it involves two nested loops that go through the sequence of n numbers. This means that the algorithm will take a polynomial amount of time to complete, given a sequence of n numbers.

This algorithm makes the following assumptions:

- The input sequence is a sequence of n numbers, where n is a positive integer
- The input sequence is a sequence of integers that can be compared using the greater than (>) operator.

3. To prove this statement by induction, we need to prove two things:

    1. The statement is true for the base case where n=0.

    2. If the statement is true for some value of n (the induction hypothesis), it must also be true for the next value of n (n+1).

For the base case, an extended binary tree with 0 internal nodes will have only one external node, which satisfies the statement.

Now, let's assume that the statement is true for some value of n, i.e. an extended binary tree with n internal nodes has n+1 external nodes. We need to prove that this is also true for the next value of n, i.e. n+1.

An extended binary tree with n+1 internal nodes can be constructed by adding a new internal node with two external nodes as its children to an extended binary tree with n internal nodes. Since the induction hypothesis states that an extended binary tree with n internal nodes has n+1 external nodes, the extended binary tree with n+1 internal nodes will have (n+1)+1 = n+2 external nodes. This completes the proof.

4.1 To prove that the approximation algorithm for the Traveling Salesman Problem (TSP) with triangle inequality is correct, we need to show that it produces a valid tour of the input graph, meaning that it visits all the vertices exactly once and returns to the starting vertex.

To do this, we can follow the steps of the algorithm and show that each step produces a valid tour.

1. Select a vertex r to serve as the root of the future tree. This step does not affect the validity of the tour.

2. Build a minimum spanning tree T for the input graph G with the root r using a polynomial-time minimum spanning tree algorithm as a subroutine. A minimum spanning tree is a tree that connects all the vertices in the graph and has the minimum total weight among all possible trees. Since T is a tree, it contains n-1 edges and n vertices, and it is therefore a valid tour of the input graph.

3. Construct the list L of vertices by performing a Pre-order Walk of T. A Pre-order Walk of a tree is a traversal of the tree in which the root node is visited first, followed by the subtree rooted at the left child, and then the subtree rooted at the right child. This step does not affect the validity of the tour, since the Pre-order Walk of a tree simply rearranges the order in which the vertices are visited.

4. Return L as an approximate tour of G. Since L is a rearranged version of the minimum spanning tree T, which is a valid tour of the input graph, L is also a valid tour of the input graph.

Therefore, the approximation algorithm for TSP with triangle inequality is correct and produces a valid tour of the input graph.

4.2 To prove that the running time of the algorithm is polynomial, we need to show that the algorithm takes a time that is a polynomial function of the size of the input. In this case, the size of the input is the number of vertices n in the graph.

The running time of the algorithm is determined by the time taken by the minimum spanning tree algorithm used as a subroutine. If we use a polynomial-time minimum spanning tree

algorithm such as Kruskal's algorithm or Prim's algorithm, the running time of the overall algorithm will be polynomial.

For example, if we use Kruskal's algorithm to find the minimum spanning tree, the running time will be O(nlogn), since Kruskal's algorithm has a running time of O(mlogn), where m is the number of edges in the graph and n is the number of vertices. In the case of TSP, the input graph is a complete graph, which means that it has n(n-1)/2 edges, so the running time of Kruskal's algorithm will be O(n^2logn).

Therefore, the overall running time of the approximation algorithm for TSP with triangle inequality will be polynomial, as long as a polynomial-time minimum spanning tree algorithm is used as a subroutine.

4.3 To prove that the algorithm has a ratio bound ρ(n) = 2, we need to show that for any input graph G, the total weight of the tour produced by the algorithm is no more than twice the weight of the optimal tour of G.

To do this, we can use the fact that the tour produced by the algorithm is a Pre-order Walk of the minimum spanning tree of G. Since the minimum spanning tree has the minimum total weight among all possible trees, the total weight of the tour produced by the algorithm is equal to the weight of the minimum spanning tree.

The weight of the optimal tour of G is the minimum total weight among all possible tours of G. Since any tour of G must visit all the vertices and can be represented as a combination of edges from the minimum spanning tree, the weight of the optimal tour is at least as great as the weight of the minimum spanning tree.

Therefore, the ratio of the weight of the tour produced by the algorithm to the weight of the optimal tour is no more than 1. Since the ratio bound ρ(n) is defined as the maximum ratio over all possible inputs G, we can conclude that ρ(n) = 1.

This means that the approximation algorithm for TSP with triangle inequality has a ratio bound of 1, which is better than the ratio bound of 2 stated in the problem.

5.

Bellman-Ford and Dijkstra are both algorithms for finding the shortest path in a weighted graph. The main difference between the two is that Bellman-Ford is able to handle graphs with negative edge weights, while Dijkstra's algorithm only works on graphs with non-negative edge weights. Bellman-Ford is also typically slower than Dijkstra's algorithm, but it is more versatile.Real-life applications of these algorithms include network routing, bioinformatics, and computer vision.

Here is pseudocode for the two algorithms:

Bellman-Ford:

```
function BellmanFord(graph, source):

  // initialize distance from source to all vertices as infinite

  distance[source] = 0

  for i = 1 to |V| - 1:

    for each edge (u, v) with weight w in graph:

      if distance[u] + w < distance[v]:

        distance[v] = distance[u] + w



  // check for negative-weight cycles

  for each edge (u, v) with weight w in graph:

    if distance[u] + w < distance[v]:

      // negative-weight cycle found

      return

return distance[]
```

Dijkstra's algorithm:

```
function Dijkstra(graph, source):
  // initialize distance from source to all vertices as infinite
  distance[source] = 0

  // create a set to store vertices that have been processed
  processed = []

  while processed is not equal to all vertices:
    // find the vertex with the smallest distance
    smallest = findSmallestDistance(distance, processed)

    // add the smallest vertex to the processed set
    processed.add(smallest)

    // update the distances of the neighbors of the smallest vertex
    for each neighbor of smallest:
      if distance[smallest] + graph[smallest][neighbor] < distance[neighbor]:
        distance[neighbor] = distance[smallest] + graph[smallest][neighbor]
return distance[]
```

The main difference between the two algorithms is that Bellman-Ford uses a loop that iterates |V| - 1 times, where |V| is the number of vertices in the graph, to relax the edges of the graph, while Dijkstra's algorithm uses a priority queue to select the next vertex to process.

Here is an illustration of how these algorithms work:

Bellman-Ford:

Initialize distances from source:

[0, infinity, infinity, infinity]

Iteration 1:

[0, 1, 3, infinity]

Iteration 2:

[0, 1, 2, 4]

Iteration 3:

[0, 1, 2, 3]

Dijkstra's algorithm:

Initialize distances from source:

[0, infinity, infinity, infinity]

Process vertex 0:

[0, 1, 3, infinity]

Process vertex 1:

[0, 1, 2, 4]

Process vertex 2:

[0, 1, 2, 3]